CHAPTER 2

Vectors and Matrices

KEY TERMS

vectors	transpose	array operations
matrices	subscripted indexing	array multiplication
row vector	unwinding a matrix	array division
column vector	linear indexing	matrix multiplication
scalar	column major order	inner dimensions
elements	columnwise	outer dimensions
array	vector of variables	dot product or inner product
array operations	empty vector	cross product or outer product
colon operator	deleting elements	logical vector
iterate	three-dimensional matrices	logical indexing
step value	cumulative sum	zero crossings
concatenating	cumulative product	
index	running sum	
subscript	nesting calls	
index vector	scalar multiplication	

CONTENTS

MATLAB® is short for matrix laboratory. Everything in MATLAB is written to work with vectors and matrices. This chapter will introduce vectors and matrices. Operations on vectors and matrices, and built-in functions that can be used to simplify code will also be explained. The matrix operations and functions described in this chapter will form the basis for vectorized coding, which will be explained in Chapter 5.

2.1 VECTORS AND MATRICES

Vectors and *matrices* are used to store sets of values, all of which are the same type. A matrix can be visualized as a table of values. The dimensions of a matrix are *r x c*, where r is the number of rows and c is the number of

MATLAB®. http://dx.doi.org/10.1016/B978-0-12-405876-7.00002-X

columns. This is pronounced "r by c". A vector can be either a ***row vector*** or a ***column vector***. If a vector has *n* elements, a row vector would have the dimensions *1 x n* and a column vector would have the dimensions *n x 1*. A ***scalar*** (one value) has the dimensions *1 x 1*. Therefore, vectors and scalars are actually just special cases of matrices.

Here are some diagrams showing, from left to right, a scalar, a column vector, a row vector, and a matrix:

5

3
7
4

5	88	3	11

9	6	3
5	7	2

The scalar is *1 x 1*, the column vector is *3 x 1* (three rows by one column), the row vector is *1 x 4* (one row by four columns), and the matrix is *2 x 3* (two rows by three columns). All of the values stored in these matrices are stored in what are called ***elements***.

MATLAB is written to work with matrices; the name MATLAB is short for matrix laboratory. As MATLAB is written to work with matrices, it is very easy to create vector and matrix variables, and there are many operations and functions that can be used on vectors and matrices.

A vector in MATLAB is equivalent to what is called a one-dimensional ***array*** in other languages. A matrix is equivalent to a two-dimensional array. Usually, even in MATLAB, some operations that can be performed on either vectors or matrices are referred to as ***array operations***. The term array is also frequently used to mean generically either a vector or a matrix.

In mathematics, the general form of an *m x n* matrix A is written as:

$$\mathrm{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} = \mathrm{a}_{ij} \quad \mathrm{i} = 1, \ldots, \mathrm{m}; \quad \mathrm{j} = 1, \ldots, \mathrm{n}$$

2.1.1 Creating Row Vectors

There are several ways to create row vector variables. The most direct way is to put the values that you want in the vector in square brackets, separated by either spaces or commas. For example, both of these assignment statements create the same vector *v*:

```
>> v = [1   2   3   4]
v =
     1   2   3   4

>> v = [1,2,3,4]
v =
     1   2   3   4
```

Both of these create a row vector variable that has four elements; each value is stored in a separate element in the vector.

2.1.1.1 The Colon Operator and Linspace Function

If, as in the preceding examples, the values in the vector are regularly spaced, the ***colon operator*** can be used to ***iterate*** through these values. For example, 1:5 results in all of the integers from 1 to 5 inclusive:

```
>> vec = 1:5
vec =
      1      2      3      4      5
```

Note that, in this case, the brackets [ ] are not necessary to define the vector.

With the colon operator, a ***step value*** can also be specified by using another colon, in the form (first:step:last). For example, to create a vector with all integers from 1 to 9 in steps of 2:

```
>> nv = 1:2:9
nv =
     1   3   5   7   9
```

QUICK QUESTION!

What happens if adding the step value would go beyond the range specified by the last, for example

```
1:2:6
```

Answer

This would create a vector containing 1, 3, and 5. Adding 2 to the 5 would go beyond 6, so the vector stops at 5; the result would be

```
      1      3      5
```

QUICK QUESTION!

How can you use the colon operator to generate the vector shown below?

```
      9      7      5      3      1
```

Answer

```
9:-2:1
```

The step value can be a negative number, so the resulting sequence is in descending order (from highest to lowest).

The **linspace** function creates a linearly spaced vector; **linspace(x,y,n)** creates a vector with *n* values in the inclusive range from *x* to *y*. If n is omitted, the default is 100 points. For example, the following creates a vector with five values linearly spaced between 3 and 15, including the 3 and 15:

```
>> ls = linspace(3,15,5)
ls =
     3     6     9    12    15
```

Similarly, the **logspace** function creates a logarithmically spaced vector; **logspace(x,y,n)** creates a vector with *n* values in the inclusive range from 10^x to 10^y. If *n* is omitted, the default is 50 points. For example:

```
>> logspace(1,5,5)
ans =
        10         100         1000        10000      100000
```

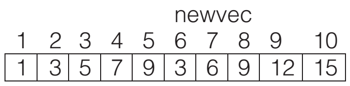
Vector variables can also be created using existing variables. For example, a new vector is created here consisting, first of all, of the values from *nv* followed by all values from *ls*:

```
>> newvec = [nv  ls]
newvec =
   1    3    5    7    9    3    6    9    12    15
```

Putting two vectors together like this to create a new one is called ***concatenating*** the vectors.

2.1.1.2 Referring to and Modifying Elements

The elements in a vector are numbered sequentially; each element number is called the ***index***, or ***subscript***. In MATLAB, the indices start at 1. Normally, diagrams of vectors and matrices show the indices. For example, for the variable *newvec* created earlier the indices 1–10 of the elements are shown above the vector:



A particular element in a vector is accessed using the name of the vector variable and the index or subscript in parentheses. For example, the fifth element in the vector *newvec* is a 9.

```
>> newvec(5)
ans =
     9
```

The expression *newvec(5)* would be pronounced "newvec sub 5", where sub is short for subscript. A subset of a vector, which would be a vector itself, can also be obtained using the colon operator. For example, the following statement would get the fourth through sixth elements of the vector *newvec*, and store the result in a vector variable *b*:

```
>> b = newvec(4:6)
b  =
    7    9    3
```

Any vector can be used for the indices into another vector, not just one created using the colon operator. The indices do not need to be sequential. For example, the following would get the first, tenth, and fifth elements of the vector *newvec*:

```
>> newvec([1 10 5])
ans =
     1     15     9
```

The vector [1 10 5] is called an ***index vector***; it specifies the indices in the original vector that are being referenced.

The value stored in a vector element can be changed by specifying the index or subscript. For example, to change the second element from the preceding vector *b* to now store the value 11 instead of 9:

```
>> b(2) = 11
b =
     7     11     3
```

By referring to an index that does not yet exist, a vector can also be extended. For example, the following creates a vector that has three elements. By then assigning a value to the fourth element, the vector is extended to have four elements.

```
>> rv = [3 55 11]
rv =
     3     55     11
>> rv(4) = 2
rv =
     3     55     11      2
```

If there is a gap between the end of the vector and the specified element, 0s are filled in. For example, the following extends the variable *rv* again:

```
>> rv(6) = 13
rv =
     3     55     11      2      0     13
```

As we will see later, this is actually not very efficient because it can take extra time.

PRACTICE 2.1

Think about what would be produced by the following sequence of statements and expressions, and then type them in to verify your answers:

```
pvec = 3:2:10

pvec(2) = 15

pvec(7) = 33

pvec([2:4  7])

linspace(5,11,3)

logspace(2,4,3)
```

2.1.2 Creating Column Vectors

One way to create a column vector is to explicitly put the values in square brackets, separated by semicolons (rather than commas or spaces):

```
>> c = [1; 2; 3; 4]
c =
     1
     2
     3
     4
```

There is no direct way to use the colon operator to get a column vector. However, any row vector created using any method can be ***transposed*** to result in a column vector. In general, the transpose of a matrix is a new matrix in which the rows and columns are interchanged. For vectors, transposing a row vector results in a column vector, and transposing a column vector results in a row vector. In MATLAB, the apostrophe is built in as the transpose operator.

```
>> r = 1:3;
>> c = r'
c =
     1
     2
     3
```

2.1.3 Creating Matrix Variables

Creating a matrix variable is simply a generalization of creating row and column vector variables. That is, the values within a row are separated by either spaces or commas, and the different rows are separated by semicolons. For example, the matrix variable *mat* is created by explicitly entering values:

```
>> mat = [4   3   1; 2   5   6]
mat =
   4   3   1
   2   5   6
```

There must always be the same number of values in each row. If you attempt to create a matrix in which there are different numbers of values in the rows, the result will be an error message, such as in the following:

```
>> mat = [3 5 7; 1 2]
Error using vertcat
Dimensions of matrices being concatenated are not consistent.
```

Iterators can be used for the values in the rows using the colon operator. For example:

```
>> mat = [2:4; 3:5]
mat =
   2   3   4
   3   4   5
```

The separate rows in a matrix can also be specified by hitting the Enter key after each row instead of typing a semicolon when entering the matrix values, as in:

```
>> newmat = [2 6 88
33 5 2]

newmat =
     2     6    88
    33     5     2
```

Matrices of random numbers can be created using the **rand** function. If a single value *n* is passed to **rand**, an *n x n* matrix will be created, or passing two arguments will specify the number of rows and columns:

```
>> rand(2)
ans =
    0.2311    0.4860
    0.6068    0.8913

>> rand(1,3)
ans =
    0.7621    0.4565    0.0185
```

Matrices of random integers can be generated using **randi**; after the range is passed, the dimensions of the matrix are passed (again, using one value *n* for an *n x n* matrix, or two values for the dimensions):

```
>> randi([5, 10], 2)
ans =
     8    10
     9     5

>> randi([10, 30], 2, 3)
ans =
    21    10    13
    19    17    26
```

Note that the range can be specified for **randi**, but not for **rand** (the format for calling these functions is different).

MATLAB also has several functions that create special matrices. For example, the **zeros** function creates a matrix of all zeros and the **ones** function creates a matrix of all ones. Like **rand**, either one argument can be passed (which will be both the number of rows and columns) or two arguments (first the number of rows and then the number of columns).

```
>> zeros(3)
ans =
     0     0     0
     0     0     0
     0     0     0

>> ones(2,4)
ans =
     1     1     1     1
     1     1     1     1
```

Note that there is no twos function, or tens, or fifty-threes — just **zeros** and **ones**!

2.1.3.1 Referring to and Modifying Matrix Elements

To refer to matrix elements, the row and then the column subscripts are given in parentheses (always the row first and then the column). For example, this creates a matrix variable *mat* and then refers to the value in the second row, third column of *mat*:

```
>> mat = [2:4; 3:5]
mat =
   2   3   4
   3   4   5

>> mat(2,3)
ans =
   5
```

This is called ***subscripted indexing***; it uses the row and column subscripts. It is also possible to refer to a subset of a matrix. For example, this refers to the first and second rows, second and third columns:

```
>> mat(1:2,2:3)
ans =
   3   4
   4   5
```

Using just one colon by itself for the row subscript means all rows, regardless of how many, and using a colon for the column subscript means all columns. For example, this refers to all columns within the first row or, in other words, the entire first row:

```
>> mat(1,:)
ans =
   2   3   4
```

This refers to the entire second column:

```
>> mat(:, 2)
ans =
   3
   4
```

If a single index is used with a matrix, MATLAB ***unwinds*** the matrix column by column. For example, for the matrix *intmat* created here, the first two elements are from the first column and the last two are from the second column:

```
>> intmat = [100 77; 28 14]
intmat =
   100    77
    28    14
>> intmat(1)
ans =
   100
>> intmat(2)
ans =
    28
>> intmat(3)
ans =
    77
>> intmat(4)
ans =
    14
```

This is called ***linear indexing***. It is usually much better style when working with matrices to use subscripted indexing.

MATLAB stores matrices in memory in ***column major order***, or ***columnwise***, which is why linear indexing refers to the elements in order by columns.

An individual element in a matrix can be modified by assigning a new value to it.

```
>> mat = [2:4; 3:5];
>> mat(1,2) = 11
mat =
       2     11      4
       3      4      5
```

An entire row or column could also be changed. For example, the following replaces the entire second row with values from a vector obtained using the colon operator.

```
>> mat(2,:) = 5:7
mat =
       2     11      4
       5      6      7
```

Notice that as the entire row is being modified, a row vector with the correct length must be assigned. Any subset of a matrix can be modified as long as what is being assigned has the same number of rows and columns as the subset being modified.

To extend a matrix an individual element could not be added as that would mean there would no longer be the same number of values in every row. However, an entire row or column could be added. For example, the following would add a fourth column to the matrix:

```
>> mat(:,4) = [9 2]'
mat =
       2     11      4      9
       5      6      7      2
```

Just as we saw with vectors, if there is a gap between the current matrix and the row or column being added, MATLAB will fill in with zeros.

```
>> mat(4,:) = 2:2:8
mat =
       2     11      4      9
       5      6      7      2
       0      0      0      0
       2      4      6      8
```

2.1.4 Dimensions

The **length** and **size** functions in MATLAB are used to find dimensions of vectors and matrices. The **length** function returns the number of elements in a vector. The **size** function returns the number of rows and columns in a vector or matrix. For example, the following vector *vec* has four elements so its length is 4. It is a row vector, so the size is *1 x 4*.

```
>> vec = -2:1
vec =
   -2   -1   0   1
>> length(vec)
ans =
   4
>> size(vec)
ans =
   1    4
```

To create the following matrix variable *mat*, iterators are used on the two rows and then the matrix is transposed so that it has three rows and two columns or, in other words, the size is *3 x 2*.

```
>> mat = [1:3; 5:7]'
mat =
   1    5
   2    6
   3    7
```

The **size** function returns the number of rows and then the number of columns, so to capture these values in separate variables we put a ***vector of two variables*** on the left of the assignment. The variable *r* stores the first value returned, which is the number of rows, and *c* stores the number of columns.

```
>> [r, c] = size(mat)
r =
  3
c =
  2
```

Note that this example demonstrates very important and unique concepts in MATLAB: the ability to have a function return multiple values and the ability to have a vector of variables on the left side of an assignment in which to store the values.

If called as just an expression, the **size** function will return both values in a vector:

```
>> size(mat)
ans =
   3    2
```

For a matrix, the **length** function will return either the number of rows or the number of columns, whichever is largest (in this case the number of rows, 3).

```
>> length(mat)
ans =
   3
```

QUICK QUESTION!

How could you create a matrix of zeros with the same size as another matrix?

Answer

For a matrix variable *mat*, the following expression would accomplish this:

```
zeros(size(mat))
```

The **size** function returns the size of the matrix, which is then passed to the **zeros** function, which then returns a matrix of zeros with the same size as *mat*. It is not necessary in this case to store the values returned from the **size** function in variables.

MATLAB also has a function **numel**, which returns the total number of elements in any array (vector or matrix):

```
>> vec = 9:-2:1
vec =
     9     7     5     3     1
>> numel(vec)
ans =
     5

>> mat = [3:2:7; 9 33 11]
mat =
     3     5     7
     9    33    11
>> numel(mat)
ans =
     6
```

For vectors, this is equivalent to the **length** of the vector. For matrices, it is the product of the number of rows and columns.

It is important to note that in programming applications, it is better to not assume that the dimensions of a vector or matrix are known. Instead, to be general, use either the **length** or **numel** function to determine the number of elements in a vector, and use **size** (and store the result in two variables) for a matrix.

MATLAB also has a built-in expression, **end**, that can be used to refer to the last element in a vector; for example, *v(end)* is equivalent to *v(length(v))*. For matrices, it can refer to the last row or column. So, for example, using **end** for the row index would refer to the last row.

In this case, the element referred to is in the first column of the last row:

```
>> mat = [1:3; 4:6]'
mat =
     1     4
     2     5
     3     6
>> mat(end,1)
ans =
     3
```

Using **end** for the column index would refer to a value in the last column (e.g., the last column of the second row):

```
>> mat(2,end)
ans =
     5
```

This can only be used as an index.

2.1.4.1 Changing Dimensions

In addition to the transpose operator, MATLAB has several built-in functions that change the dimensions or configuration of matrices, including **reshape**, **fliplr**, **flipud**, and **rot90**.

The **reshape** function changes the dimensions of a matrix. The following matrix variable *mat* is *3 x 4* or, in other words, it has 12 elements (each in the range from 1 to 100).

```
>> mat = randi(100, 3, 4)
    14    61     2    94
    21    28    75    47
    20    20    45    42
```

These 12 values could instead be arranged as a *2 x 6* matrix, *6 x 2*, *4 x 3*, *1 x 12*, or *12 x 1*. The **reshape** function iterates through the matrix columnwise. For example, when reshaping *mat* into a *2 x 6* matrix, the values from the first column in the original matrix (14, 21, and 20) are used first, then the values from the second column (61, 28, 20), and so forth.

```
>> reshape(mat,2,6)
ans =
    14    20    28     2    45    47
    21    61    20    75    94    42
```

Note that in these examples *mat* is unchanged; instead, the results are stored in the default variable *ans* each time.

The **fliplr** function "flips" the matrix from left to right (in other words, the left-most column, the first column, becomes the last column and so forth), and the **flipud** function flips up to down.

```
>> mat
mat =
    14    61     2    94
    21    28    75    47
    20    20    45    42
>> fliplr(mat)
ans =
    94     2    61    14
    47    75    28    21
    42    45    20    20
```

```
>> mat
mat =
     14     61      2     94
     21     28     75     47
     20     20     45     42
>> flipud(mat)
ans =
     20     20     45     42
     21     28     75     47
     14     61      2     94
```

The **rot90** function rotates the matrix counterclockwise 90 degrees, so, for example, the value in the top right corner becomes instead the top left corner and the last column becomes the first row.

```
>> mat
mat =
     14     61      2     94
     21     28     75     47
     20     20     45     42
>> rot90(mat)
ans =
     94     47     42
      2     75     45
     61     28     20
     14     21     20
```

QUICK QUESTION!

Is there a ***rot180*** function? Is there a ***rot90*** function (to rotate clockwise)?

Answer

Not exactly, but a second argument can be passed to the **rot90** function which is an integer n; the function will rotate 90*n degrees. The integer can be positive or negative. For example, if 2 is passed, the function will rotate the matrix 180 degrees (so, it would be the same as rotating the result of **rot90** another 90 degrees).

```
>> mat
mat =
     14     61      2     94
     21     28     75     47
     20     20     45     42
>> rot90(mat,2)
ans =
     42     45     20     20
     47     75     28     21
     94      2     61     14
```

If a negative number is passed for n, the rotation would be in the opposite direction, that is, clockwise.

```
>> mat
mat =
     14     61      2     94
     21     28     75     47
     20     20     45     42
>> rot90(mat,-1)
ans =
     20     21     14
     20     28     61
     45     75      2
     42     47     94
```

The function **repmat** can be used to create a matrix; **repmat(mat,m,n)** creates a larger matrix that consists of an *m x n* matrix of copies of *mat*. For example, here is a *2 x 2* random matrix:

```
>> intmat = randi(100,2)
intmat =
    50     34
    96     59
```

Replicating this matrix six times as a *3 x 2* matrix would produce copies of *intmat* in this form:

intmat	intmat
intmat	intmat
intmat	intmat

```
>> repmat(intmat,3,2)
ans =
    50     34     50     34
    96     59     96     59
    50     34     50     34
    96     59     96     59
    50     34     50     34
    96     59     96     59
```

2.1.5 Empty Vectors

An ***empty vector*** (a vector that stores no values) can be created using empty square brackets:

```
>> evec = []
evec =
     []
>> length(evec)
ans =
   0
```

Note
There is a difference between having an empty vector variable and not having the variable at all.

Values can then be added to an empty vector by concatenating, or adding, values to the existing vector. The following statement takes what is currently in *evec*, which is nothing, and adds a 4 to it.

```
>> evec = [evec 4]
evec =
     4
```

The following statement takes what is currently in *evec*, which is 4, and adds an 11 to it.

```
>> evec = [evec 11]
evec =
     4     11
```

This can be continued as many times as desired to build a vector up from nothing. Sometimes this is necessary, although, generally, it is not a good idea if it can be avoided because it can be quite time consuming.

Empty vectors can also be used to ***delete elements*** from vectors. For example, to remove the third element from a vector, the empty vector is assigned to it:

```
>> vec = 4:8
vec =
     4      5      6      7      8
>> vec(3) = []
vec =
     4      5      7      8
```

The elements in this vector are now numbered 1 through 4.

Subsets of a vector could also be removed. For example:

```
>> vec = 3:10
vec =
     3      4      5      6      7      8      9     10
>> vec(2:4) = []
vec =
     3      7      8      9     10
```

Individual elements cannot be removed from matrices, as matrices always have to have the same number of elements in every row.

```
>> mat = [7 9 8; 4 6 5]
mat =
     7      9      8
     4      6      5
>> mat(1,2) = [];
Subscripted assignment dimension mismatch.
```

However, entire rows or columns could be removed from a matrix. For example, to remove the second column:

```
>> mat(:,2) = []
mat =
     7      8
     4      5
```

Also, if linear indexing is used with a matrix to delete an element, the matrix will be reshaped into a row vector.

```
>> mat = [7 9 8; 4 6 5]
mat =
     7     9     8
     4     6     5
>> mat(3) = []
mat =
     7     4     6     8     5
```

PRACTICE 2.2

Think about what would be produced by the following sequence of statements and expressions, and then type them in to verify your answers.

```
mat = [1:3; 44 9  2; 5:-1:3]

mat(3,2)

mat(2,:)

size(mat)

mat(:,4) = [8;11;33]

numel(mat)

v = mat(3,:)

v(v(2))

v(1) = []

reshape(mat,2,6)
```

2.1.6 Three-Dimensional Matrices

The matrices that have been shown so far have been two-dimensional; these matrices have rows and columns. Matrices in MATLAB are not limited to two dimensions, however. In fact, in Chapter 13 we will see image applications in which ***three-dimensional matrices*** are used. For a three-dimensional matrix, imagine a two-dimensional matrix as being flat on a page, and then the third dimension consists of more pages on top of that one (so they are stacked on top of each other).

Here is an example of creating a three-dimensional matrix. First, two two-dimensional matrices *layerone* and *layertwo* are created; it is important that they have the same dimensions (in this case, *3 x 5*). Then, these are made into "layers" in a three-dimensional matrix *mat*. Note that we end up with a matrix that has two layers, each of which is *3 x 5*. The resulting three-dimensional matrix has dimensions *3 x 5 x 2*.

```
>> layerone = reshape(1:15,3,5)
layerone =
      1      4      7     10     13
      2      5      8     11     14
      3      6      9     12     15
>> layertwo = fliplr(flipud(layerone))
layertwo =
     15     12      9      6      3
     14     11      8      5      2
     13     10      7      4      1

>> mat(:,:,1) = layerone
mat =
      1      4      7     10     13
      2      5      8     11     14
      3      6      9     12     15
>> mat(:,:,2) = layertwo
mat(:,:,1) =
      1      4      7     10     13
      2      5      8     11     14
      3      6      9     12     15
mat(:,:,2) =
     15     12      9      6      3
     14     11      8      5      2
     13     10      7      4      1
>> size(mat)
ans =
      3      5      2
```

Three-dimensional matrices can also be created using the **zeros**, **ones**, and **rand** functions by specifying three dimensions to begin with. For example, **zeros(2,4,3)** will create a *2 x 4 x 3* matrix of all 0s.

Unless specified otherwise, in the remainder of this book "matrices" will be assumed to be two-dimensional.

2.2 VECTORS AND MATRICES AS FUNCTION ARGUMENTS

In MATLAB an entire vector or matrix can be passed as an argument to a function; the function will be evaluated on every element. This means that the result will be the same size as the input argument.

For example, let us find the sine in radians of every element of a vector *vec*. The **sin** function will automatically return the sine of each individual element and the result will be a vector with the same length as the input vector.

```
>> vec = -2:1
vec =
    -2     -1      0      1
>> sinvec = sin(vec)
sinvec =
   -0.9093   -0.8415         0    0.8415
```

For a matrix, the resulting matrix will have the same size as the input argument matrix. For example, the **sign** function will find the sign of each element in a matrix:

```
>> mat = [0 4 -3; -1 0 2]
mat =
     0      4     -3
    -1      0      2
>> sign(mat)
ans =
     0      1     -1
    -1      0      1
```

Functions such as **sin** and **sign** can have either scalars or arrays (vectors or matrices) passed to them. There are a number of functions that are written specifically to operate on vectors or on columns of matrices; these include the functions **min**, **max**, **sum**, **prod**, **cumsum**, and **cumprod**. These functions will be demonstrated first with vectors and then with matrices.

For example, assume that we have the following vector variables:

```
>> vec1 = 1:5;
>> vec2 = [3 5 8 2];
```

The function **min** will return the minimum value from a vector, and the function **max** will return the maximum value.

```
>> min(vec1)
ans =
     1
>> max(vec2)
ans =
     8
```

The function **sum** will sum all of the elements in a vector. For example, for *vec1* it will return 1+2+3+4+5 or 15:

```
>> sum(vec1)
ans =
    15
```

The function **prod** will return the product of all of the elements in a vector; for example, for *vec2* it will return 3*5*8*2 or 240:

```
>> prod(vec2)
ans =
   240
```

The functions **cumsum** and **cumprod** return the ***cumulative sum*** or ***cumulative product***, respectively. A cumulative, or ***running sum***, stores the sum so far at each step as it adds the elements from the vector. For example, for *vec1*, it would store the first element, 1, then 3 (1+2), then 6 (1+2+3), then 10 (1+2+3+4), then, finally, 15 (1+2+3+4+5). The result is a vector that has as many elements as the input argument vector that is passed to it:

```
>> cumsum(vec1)
ans =
     1     3     6    10    15
>> cumsum(vec2)
ans =
     3     8    16    18
```

The **cumprod** function stores the cumulative products as it multiplies the elements in the vector together; again, the resulting vector will have the same length as the input vector:

```
>> cumprod(vec1)
ans =
     1     2     6    24   120
```

For matrices, all of these functions operate on every individual column. If a matrix has dimensions *r x c*, the result for the **min**, **max**, **sum**, and **prod** functions will be a *1 x c* row vector, as they return the minimum, maximum, sum, or product, respectively, for every column. For example, assume the following matrix:

```
>> mat = randi([1 20], 3, 5)
mat =
     3    16     1    14     8
     9    20    17    16    14
    19    14    19    15     4
```

The following are the results for the **max** and **sum** functions:

```
>> max(mat)
ans =
    19    20    19    16    14
>> sum(mat)
ans =
    31    50    37    45    26
```

To find a function for every row, instead of every column, one method would be to transpose the matrix.

```
>> max(mat')
ans =
    16    20    19
>> sum(mat')
ans =
    42    76    71
```

As columns are the default, they are considered to be the first dimension. Specifying the second dimension as an argument to one of these functions will result in the function operating rowwise. The syntax is slightly different; for the **sum** and **prod** functions, this is the second argument, whereas for the **min** and **max** functions it must be the third argument and the second argument must be an empty vector:

```
>> max(mat,[],2)
ans =
    16
    20
    19
>> sum(mat,2)
ans =
    42
    76
    71
```

Note the difference in the format of the output with these two methods (transposing results in row vectors whereas specifying the second dimension results in column vectors).

QUICK QUESTION!

As these functions operate columnwise, how can we get an overall result for the matrix? For example, how would we determine the overall maximum in the matrix?

Answer

We would have to get the maximum from the row vector of column maxima, in other words ***nest the calls*** to the **max** function:

```
>> max(max(mat))
ans =
    20
```

For the **cumsum** and **cumprod** functions, again they return the cumulative sum or product of every column. The resulting matrix will have the same dimensions as the input matrix:

```
>> mat
mat =
     3    16     1    14     8
     9    20    17    16    14
    19    14    19    15     4
>> cumsum(mat)
ans =
     3    16     1    14     8
    12    36    18    30    22
    31    50    37    45    26
```

2.3 SCALAR AND ARRAY OPERATIONS ON VECTORS AND MATRICES

Numerical operations can be done on entire vectors or matrices. For example, let's say that we want to multiply every element of a vector *v* by 3.

In MATLAB, we can simply multiply *v* by 3 and store the result back in *v* in an assignment statement:

```
>> v = [3 7 2 1];
>> v = v*3
v =
     9    21     6     3
```

As another example, we can divide every element by 2:

```
>> v = [3 7 2 1];
>> v/2
ans =
    1.5000    3.5000    1.0000    0.5000
```

To multiply every element in a matrix by 2:

```
>> mat = [4:6; 3:-1:1]
mat =
     4     5     6
     3     2     1
>> mat * 2
ans =
     8    10    12
     6     4     2
```

This operation is referred to as ***scalar multiplication***. We are multiplying every element in a vector or matrix by a scalar (or dividing every element in a vector or a matrix by a scalar).

QUICK QUESTION!

There is no tens function to create a matrix of all tens, so how could we accomplish that?

Answer

We can either use the **ones** function and multiply by ten, or the **zeros** function and add ten:

```
>> ones(1,5) * 10
ans =
    10    10    10    10    10
>> zeros(2) + 10
ans =
    10    10
    10    10
```

Array operations are operations that are performed on vectors or matrices term by term or element by element. This means that the two arrays (vectors or matrices) must be the same size to begin with. The following examples demonstrate the array addition and subtraction operators.

```
>> v1 = 2:5
v1 =
     2     3     4     5
>> v2 = [33 11 5 1]
v2 =
    33    11     5     1
>> v1 + v2
ans =
    35    14     9     6

>> mata = [5:8; 9:-2:3]
mata =
     5     6     7     8
     9     7     5     3
>> matb = reshape(1:8,2,4)
matb =
     1     3     5     7
     2     4     6     8
>> mata - matb
ans =
     4     3     2     1
     7     3    -1    -5
```

However, for any operation that is based on multiplication (which means multiplication, division, and exponentiation), a dot must be placed in front of the operator for array operations. For example, for the exponentiation operator .^ must be used when working with vectors and matrices, rather than just the ^ operator. Squaring a vector, for example, means multiplying each element by itself so the .^ operator must be used.

```
>> v = [3 7 2 1];
>> v ^ 2
Error using  ^
Inputs must be a scalar and a square matrix.
To compute elementwise POWER, use POWER (.^) instead.

>> v .^ 2
ans =
     9    49     4     1
```

Similarly, the operator .* must be used for **array multiplication** and ./ or .\ for **array division**. The following examples demonstrate array multiplication and array division.

```
>> v1 = 2:5
v1 =
     2     3     4     5
>> v2 = [33 11 5 1]
v2 =
    33    11     5     1

>> v1 .* v2
ans =
    66    33    20     5

>> mata = [5:8; 9:-2:3]
mata =
     5     6     7     8
     9     7     5     3
>> matb = reshape(1:8, 2,4)
matb =
     1     3     5     7
     2     4     6     8

>> mata ./ matb
ans =
    5.0000    2.0000    1.4000    1.1429
    4.5000    1.7500    0.8333    0.3750
```

The operators .^, .*, ./, and .\ are called array operators and are used when multiplying or dividing vectors or matrices of the same size term by term. Note that matrix multiplication is a very different operation, and will be covered in the next section.

PRACTICE 2.3

Create a vector variable and subtract 3 from every element in it.
Create a matrix variable and divide every element by 3.
Create a matrix variable and square every element.

2.4 MATRIX MULTIPLICATION

Matrix multiplication does *not* mean multiplying term by term; it is not an array operation. Matrix multiplication has a very specific meaning. First of all, to multiply a matrix A by a matrix B to result in a matrix C, the number of columns of A must be the same as the number of rows of B. If the matrix A has dimensions $m\ x\ n$, that means that matrix B must have dimensions $n\ x$ *something*; we'll call it p.

We say that the ***inner dimensions*** (the ns) must be the same. The resulting matrix C has the same number of rows as A and the same number of columns as B (i.e., the ***outer dimensions*** $m\ x\ p$). In mathematical notation,

$$[\mathrm{A}]_{m\ x\ n}\ [\mathrm{B}]_{n\ x\ p} = [\mathrm{C}]_{m\ x\ p}$$

This only defines the size of C, not how to find the elements of C.

The elements of the matrix C are defined as the sum of products of corresponding elements in the rows of A and columns of B, or, in other words

$$\mathrm{c_{ij}}\ \sum_{k=1}^{n} a_{ik} b_{kj}.$$

In the following example, A is $2\ x\ 3$ and B is $3\ x\ 4$; the inner dimensions are both 3, so performing the matrix multiplication A*B is possible (note that B*A would not be possible). C will have as its size the outer dimensions $2\ x\ 4$. The elements in C are obtained using the summation just described. The first row of C is obtained using the first row of A and in succession the columns of B. For example, C(1,1) is 3*1 +8*4+0*0 or 35. C(1,2) is 3*2+8*5+0*2 or 46.

$$\overset{\text{A}}{\begin{bmatrix} 3 & 8 & 0 \\ 1 & 2 & 5 \end{bmatrix}} * \overset{\text{B}}{\begin{bmatrix} 1 & 2 & 3 & 1 \\ 4 & 5 & 1 & 2 \\ 0 & 2 & 3 & 0 \end{bmatrix}} = \overset{\text{C}}{\begin{bmatrix} 35 & 46 & 17 & 19 \\ 9 & 22 & 20 & 5 \end{bmatrix}}$$

In MATLAB, the * operator will perform this matrix multiplication:

```
>> A = [3 8 0; 1 2 5];
>> B = [1 2 3 1; 4 5 1 2; 0 2 3 0];
>> C = A*B
C =
     35     46     17     19
      9     22     20      5
```

PRACTICE 2.4

When two matrices have the same dimensions and are square, both array and matrix multiplication can be performed on them. For the following two matrices perform A.*B, A*B, and B*A by hand and then verify the results in MATLAB.

$$\overset{\text{A}}{\begin{bmatrix} 1 & 4 \\ 3 & 3 \end{bmatrix}} \quad \overset{\text{B}}{\begin{bmatrix} 1 & 2 \\ -1 & 0 \end{bmatrix}}$$

2.4.1 Matrix Multiplication for Vectors

As vectors are just special cases of matrices, the matrix operations described previously (addition, subtraction, scalar multiplication, multiplication, transpose) also work on vectors, as long as the dimensions are correct.

For vectors, we have already seen that the transpose of a row vector is a column vector, and the transpose of a column vector is a row vector.

To multiply vectors, they must have the same number of elements, but one must be a row vector and the other a column vector. For example, for a column vector *c* and row vector *r*:

$$c = \begin{bmatrix} 5 \\ 3 \\ 7 \\ 1 \end{bmatrix} \qquad r = [6\ \ 2\ \ 3\ \ 4]$$

Note that r is *1 x 4*, and c is *4 x 1*, so

$$[r]_{1\ x\ 4}[c]_{4\ x\ 1} = [s]_{1\ x\ 1}$$

or, in other words, a scalar:

$$[6\ \ 2\ \ 3\ \ 4]\begin{bmatrix} 5 \\ 3 \\ 7 \\ 1 \end{bmatrix} = 6^*5 + 2^*3 + 3^*7 + 4^*1 = 61$$

whereas $[c]_{4\ x\ 1}\ [r]_{1\ x\ 4} = [M]_{4\ x\ 4}$, or in other words a *4 x 4* matrix:

$$\begin{bmatrix} 5 \\ 3 \\ 7 \\ 1 \end{bmatrix}[6\ \ 2\ \ 3\ \ 4] = \begin{bmatrix} 30 & 10 & 15 & 20 \\ 18 & 6 & 9 & 12 \\ 42 & 14 & 21 & 28 \\ 6 & 2 & 3 & 4 \end{bmatrix}$$

In MATLAB, these operations are accomplished using the * operator, which is the matrix multiplication operator. First, the column vector *c* and row vector *r* are created.

```
>> c = [5 3 7 1]';
>> r = [6 2 3 4];
>> r*c
ans =
    61

>> c*r
ans =
    30    10    15    20
    18     6     9    12
    42    14    21    28
     6     2     3     4
```

There are also operations specific to vectors: the ***dot product*** and ***cross product***. The ***dot* product**, or ***inner product***, of two vectors a and b is written as a • b and is defined as

$$a_1b_1 + a_2b_2 + a_3b_3 + \ldots + a_nb_n = \sum_{i=1}^{n} a_ib_i$$

where both a and b have n elements, and a_i and b_i represent elements in the vectors. In other words, this is like matrix multiplication when multiplying a row vector *a* by a column vector *b*; the result is a scalar. This can be accomplished using the * operator and transposing the second vector, or by using the **dot** function in MATLAB:

```
>> vec1 = [4 2 5 1];
>> vec2 = [3 6 1 2];
>> vec1*vec2'
ans =
     31

>> dot(vec1,vec2)
ans =
     31
```

The ***cross product*** or ***outer product*** *a x b* of two vectors *a* and *b* is defined only when both a and b have three elements. It can be defined as a matrix multiplication of a matrix composed from the elements from *a* in a particular manner shown here and the column vector *b*.

$$a \times b = \begin{bmatrix} 0 & -a_3 & a_2 \\ a_3 & 0 & -a_1 \\ -a_2 & a_1 & 0 \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} = [a_2b_3 - a_3b_2, a_3b_1 - a_1b_3, a_1b_2 - a_2b_1]$$

MATLAB has a built-in function **cross** to accomplish this.

```
>> vec1 = [4 2 5];
>> vec2 = [3 6 1];
>> cross(vec1,vec2)
ans =
    -28     11     18
```

2.5 LOGICAL VECTORS

Logical vectors use relational expressions that result in **true**/**false** values.

2.5.1 Relational Expressions with Vectors and Matrices

Relational operators can be used with vectors and matrices. For example, let's say that there is a vector *vec*, and we want to compare every element in the vector to 5 to determine whether it is greater than 5 or not. The result would be a vector (with the same length as the original) with **logical true** or **false** values.

```
>> vec = [5 9 3 4 6 11];
>> isg = vec > 5
isg =
     0     1     0     0     1     1
```

Note that this creates a vector consisting of all **logical true** or **false** values. Although the result is a vector of ones and zeros, and numerical operations can be done on the vector *isg*, its type is **logical** rather than **double**.

```
>> doubres = isg + 5
doubres =
     5     6     5     5     6     6

>> whos
  Name       Size            Bytes  Class

  doubres    1x6                48  double array
  isg        1x6                 6  logical array
  vec        1x6                48  double array
```

To determine how many of the elements in the vector *vec* were greater than 5, the **sum** function could be used on the resulting vector *isg*:

```
>> sum(isg)
ans =
     3
```

What we have done is to create a ***logical vector*** *isg*. This logical vector can be used to index into the original vector. For example, if only the elements from the vector that are greater than 5 are desired:

```
>> vec(isg)
ans =
     9     6    11
```

This is called ***logical indexing***. Only the elements from *vec* for which the corresponding element in the logical vector *isg* is **logical true** are returned.

QUICK QUESTION!

Why doesn't the following work?

```
>> vec = [5 9 3 4 6 11];
>> v = [0 1 0 0 1 1];
>> vec(v)
Subscript indices must either be real
  positive integers or logicals.
```

Answer

The difference between the vector in this example and *isg* is that *isg* is a vector of logicals (**logical** 1s and 0s), whereas [0 1 0 0 1 1] by default is a vector of **double** values. ***Only logical 1s and 0s can be used to index into a vector.*** So, type casting the variable *v* would work:

```
>> v = logical(v);
>> vec(v)
ans =
     9     6    11
```

To create a vector or matrix of all **logical** 1s or 0s, the functions **true** and **false** can be used.

```
>> false(2)
ans =
     0     0
     0     0
>> true(1,5)
ans =
     1     1     1     1     1
```

The functions **true** and **false** and are faster and manage memory more efficiently than using **logical** with **zeros** or **ones**.

2.5.2 Logical Built-in Functions

There are built-in functions in MATLAB, which are useful in conjunction with **logical** vectors or matrices; two of these are the functions **any** and **all**. The function **any** returns **logical true** if any element in a vector represents **true**, and **false** if not. The function **all** returns **logical true** only if all elements represent **true**. Here are some examples.

```
>> any(isg)
ans =
     1
>> all(true(1,3))
ans =
     1
```

For the following variable *vec2*, some, but not all, elements are **true**; consequently, **any** returns **true** but **all** returns **false**.

```
>> vec2 = logical([1 1 0 1])
vec2 =
     1     1     0     1
>> any(vec2)
ans =
     1
>> all(vec2)
ans =
     0
```

The function **find** returns the indices of a vector that meet given criteria. For example, to find all of the elements in a vector that are greater than 5:

```
>> vec = [5 3 6 7 2]
vec =
     5     3     6     7     2
>> find(vec > 5)
ans =
     3     4
```

For matrices, the **find** function will use linear indexing when returning the indices that meet the specified criteria. For example:

```
>> mata = randi(10,2,4)
mata =
     5     6     7     8
     9     7     5     3
>> find(mata == 5)
ans =
     1
     6
```

For both vectors and matrices, an empty vector will be returned if no elements match the criterion. For example,

```
>> find(mata == 11)
ans =
   Empty matrix: 0-by-1
```

The function **isequal** is useful in comparing arrays. In MATLAB, using the equality operator with arrays will return 1 or 0 for each element; the **all** function could then be used on the resulting array to determine whether all elements were equal or not. The built-in function **isequal** also accomplishes this:

```
>> vec1 = [1 3 -4 2 99];
>> vec2 = [1 2 -4 3 99];
>> vec1 == vec2
ans =
     1     0     1     0     1
>> all(vec1 == vec2)
ans =
     0
>> isequal(vec1,vec2)
ans =
     0
```

However, one difference is that if the two arrays are not the same dimensions, the **isequal** function will return **logical** 0, whereas using the equality operator will result in an error message.

QUICK QUESTION!

If we have a vector *vec* that erroneously stores negative values, how can we eliminate those negative values?

Answer

One method is to determine where they are and delete these elements:

```
>> vec = [11 -5 33 2 8 -4 25];
>> neg = find(vec < 0)
neg =
     2     6
```

```
>> vec(neg) = []
vec =
    11    33     2     8    25
```

Alternatively, we can just use a logical vector rather than **find**:

```
>> vec = [11 -5 33 2 8 -4 25];
>> vec(vec < 0) = []
vec =
    11    33     2     8    25
```

PRACTICE 2.5

Modify the result seen in the previous *Quick Question!*. Instead of deleting the "bad" elements, retain only the "good" ones. (Hint: do it two ways, using **find** and using a logical vector with the expression *vec >= 0*.)

MATLAB also has **or** and **and** operators that work elementwise for arrays:

Operator	Meaning
|	elementwise or for arrays
&	elementwise and for arrays

These operators will compare any two vectors or matrices, as long as they are the same size, element by element, and return a vector or matrix of the same size of **logical** 1s and 0s. The operators || and && are only used with scalars, not matrices. For example:

```
>> v1 = logical([1 0 1 1]);
>> v2 = logical([0 0 1 0]);

>> v1 & v2
ans =
     0     0     1     0

>> v1 | v2
ans =
     1     0     1     1

>> v1 && v2
Operands to the || and && operators must be convertible to logical
scalar values.
```

As with the numerical operators, it is important to know the operator precedence rules. Table 2.1 shows the rules for the operators that have been covered so far, in the order of precedence.

Table 2.1 Operator Precedence Rules

Operators	Precedence
parentheses: ( )	Highest
transpose and power: ', ^, .^	
unary: negation (-), not (~)	
multiplication, division *,/,\\,.*,./,.\\	
addition, subtraction +, -	
relational <, <=, >, >=, ==, ~=	
element-wise and &	
element-wise or |	
and && (scalars)	
or || (scalars)	
assignment =	Lowest

2.6 APPLICATIONS: THE DIFF AND MESHGRID FUNCTIONS

Two functions that can be useful in working with applications of vectors and matrices include **diff** and **meshgrid**. The function **diff** returns the differences between consecutive elements in a vector. For example,

```
>> diff([4 7 15 32])
ans =
      3     8    17

>> diff([4 7 2 32])
ans =
      3    -5    30
```

For a vector *v* with a length of *n*, the length of **diff(v)** will be $n - 1$. For a matrix, the **diff** function will operate on each column.

```
>> mat = randi(20, 2,3)
mat =
    17     3    13
    19    19     2
>> diff(mat)
ans =
      2    16   -11
```

As an example, a vector that stores a signal can contain both positive and negative values. (For simplicity, we will assume no zeros, however.) For many applications it is useful to find the ***zero crossings***, or where the signal goes from being positive to negative or vice versa. This can be accomplished using the functions **sign**, **diff**, and **find**.

```
>> vec = [0.2 -0.1 -0.2 -0.1 0.1 0.3 -0.2];
>> sv = sign(vec)
sv =
     1    -1    -1    -1     1     1    -1

>> dsv = diff(sv)
dsv =
    -2     0     0     2     0    -2

>> find(dsv ~= 0)
ans =
     1     4     6
```

This shows that the signal crossings are between elements 1 and 2, 4 and 5, and 6 and 7.

The **meshgrid** function can specify the x and y coordinates of points in images, or can be used to calculate functions on two variables *x* and *y*. It receives as input arguments two vectors, and returns as output arguments two matrices that specify separately x and y values. For example, the x and y coordinates of a *2 x 3* image would be specified by the coordinates:

```
(1,1) (2,1) (3,1)
(1,2) (2,2) (3,2)
```

The matrices that separately specify the coordinates are created by the **meshgrid** function, where x iterates from 1 to 3 and y iterates from 1 to 2:

```
>> [x y] = meshgrid(1:3,1:2)
x =
     1     2     3
     1     2     3
y =
     1     1     1
     2     2     2
```

As another example, let's say we want to evaluate a function *f* of two variables *x* and *y*:

```
f(x,y) = 2*x + y
```

where x ranges from 1 to 4 and y ranges from 1 to 3. We can accomplish this by creating x and y matrices using **meshgrid**, and then the expression to calculate f uses scalar multiplication and array addition.

```
>> [x y] = meshgrid(1:4,1:3)
x =
      1     2     3     4
      1     2     3     4
      1     2     3     4
y =
      1     1     1     1
      2     2     2     2
      3     3     3     3

>> f = 2*x + y
f =
      3     5     7     9
      4     6     8    10
      5     7     9    11
```

Explore Other Interesting Features

- There are many functions that create special matrices (e.g., **hilb** for a Hilbert matrix, **magic**, and **pascal**).
- The **gallery** function, which can return many different types of test matrices for problems.
- The **ndims** function to find the number of dimensions of an argument.
- The **shiftdim** function.
- The **circshift** function. How can you get it to shift a row vector, resulting in another row vector?
- How to reshape a three-dimensional matrix.
- Passing three-dimensional matrices to functions. For example, if you pass a *3 x 5 x 2* matrix to the **sum** function, what would be the size of the result?

Summary

Common Pitfalls

- Attempting to create a matrix that does not have the same number of values in each row.
- Confusing matrix multiplication and array multiplication. Array operations, including multiplication, division, and exponentiation, are performed term by term (so the arrays must have the same size); the operators are .*, ./, .\, and .^. For matrix multiplication to be possible, the inner dimensions must agree and the operator is *.

- Attempting to use an array of **double** 1s and 0s to index into an array (must be **logical**, instead).
- Forgetting that for array operations based on multiplication the dot must be used in the operator. In other words, for multiplying, dividing by, dividing into, or raising to an exponent term by term, the operators are .*, ./, .\, and .^.
- Attempting to use || or && with arrays. Always use | and & when working with arrays; || and && are only used with scalars.

Programming Style Guidelines

- If possible, try not to extend vectors or matrices, as it is not very efficient.
- Do not use just a single index when referring to elements in a matrix; instead, use both the row and column subscripts (use subscripted indexing rather than linear indexing).
- To be general, never assume that the dimensions of any array (vector or matrix) are known. Instead, use the function **length** or **numel** to determine the number of elements in a vector, and the function **size** for a matrix:

```
len = length(vec);

[r, c] = size(mat);
```

- Use **true** instead of **logical(1)** and **false** instead of **logical(0)**, especially when creating vectors or matrices.

MATLAB Functions and Commands

linspace	end	max	any
logspace	reshape	sum	all
zeros	fliplr	prod	find
ones	flipud	cumsum	isequal
length	rot90	cumprod	diff
size	repmat	dot	meshgrid
numel	min	cross	

MATLAB Operators

colon :	matrix multiplication *
transpose ’	elementwise or for matrices |
array operators .^, .*, ./, .\	elementwise and for matrices &